Dependence Analysis for Loop-Level Parallelism

- For shared-memory parallel processing, we have a few options:
  - Write a “clean” sequential program as an operational definition of the computational task
    - Need compiler analysis of the program semantics and dependences to generate a parallel version
  - Write a sequential program annotated by parallelization directives, e.g. OpenMP pragmas
  - Write an explicit parallel program using threads or processes
- For OpenMP or explicit parallel programs, we need compiler analysis to analyze parallel program semantics and dependences (more difficult than sequential code, in some sense)

Dependence Analysis on Loops in Sequential Programs

- The goal is to execute different iterations in parallel
- What are the constraints?
- Control dependences
- Whether variables can be “privatized”
- Loop-carried data dependences due to shared variables

Control Dependences

- Unless we know the number of iterations in advance (no later than when the loop starts), we need to postpone the start of a new iteration until certain conditions are satisfied
  - In the next example, where is the earliest decision point?

?

```
while node {
   // process node;
   node->data1 = …
   node->data2 = …
   node = node->next;
}
```

```
Tmp = node;
Tmp->data1 ..
Tmp->data2 ..
```

Control Dependences (Cont'd)

- If a loop has many possible exits, control dependence can be more subtle

```
while continue {
    …
    if z continue = true
    else continue = false;

    …
    if x break;
    …
    if y break;
    …
}
```

Speculative iterations

- One can optimistically jump start an iteration
- Squash the iteration it turns out to be pre-mature
  - Was there any “visible” impact which must be undone?

```
while continue {
…
…
node = node->next;
…
Count++;
}
```

If “node” is a dead value at the end of the loop, but count is not, then the jump started iteration can safely continue up to the point of “Count++” operation

Loop-Carried Dependence

(Some of the slides are borrowed from Engelen at FSU)

```
   for (i = 0; i < N; i++)
S1   a[i+1] = a[i] + b[i]
```

```
S1 a[1] = a[0] + b[0]
S1 a[2] = a[1] + b[1]
S1 a[3] = a[2] + b[2]
     …
```

- Statement S_1 in iteration i has a flow dependence on S_1 in iteration i+1
- We also say that S_1 has a flow dependence on itself that is carried by loop i.
- Or simply, S_1 has a loop carried flow dependence on itself.
- Or even simpler, the i loop has a loop-carried dependence.

Iteration Vector

- Definition
  - Given a nest of n loops, the *iteration vector* $\boldsymbol{i}$ of a particular iteration of the innermost loop is a vector of integers
    $\boldsymbol{i} = (i_1, i_2, \ldots, i_n)$
    where i_k represents the iteration number for the loop at nesting level k
  - The set of all possible iteration vectors is an *iteration space*

Iteration Vector Ordering

- The iteration vectors are naturally ordered according to a *lexicographical order*, e.g. iteration (1,2) precedes (2,1) and (2,2) in the example on the previous slide
- Definition
  - Iteration $\boldsymbol{i}$ *precedes* iteration $\boldsymbol{j}$, denoted $\boldsymbol{i} < \boldsymbol{j}$, iff
    1) $\boldsymbol{i}[1{:}n\text{-}1] < \boldsymbol{j}[1{:}n\text{-}1]$, or
    2) $\boldsymbol{i}[1{:}n\text{-}1] = \boldsymbol{j}[1{:}n\text{-}1]$ and $i_n < j_n$

Loop Dependence

- Definition
  - There exist a dependence from S_1 to S_2 in a loop nest iff there exist two iteration vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ such that

    1) $\boldsymbol{i} < \boldsymbol{j}$ and there is a path from S_1 to S_2

    2) S_1 accesses memory location M on iteration $\boldsymbol{i}$ and S_2 accesses memory location M on iteration $\boldsymbol{j}$

    3) one of these accesses is a write

An Example

```
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++
S1      a[i][j] = a[j][i]
```

- Does either the i loop or the j loop have loop-carried dependences?

Dependence Testing

- Assuming affine subscripts :
  $a_1 i_1 + a_2 i_2 + \ldots + a_n i_n + e$
- Begin with single-dimension arrays.

Dependence Equation

```
    for(i=0;i<N,i++)
S1    a[f(i)] = a[g(i)]
```

- A *dependence equation* defines the access requirement

To prove **flow dependence**: for which values of $\alpha < \beta$ is

f(α) **=** **g**(β)

```
    for(i=0;i<N,i++)
S1    a[i+1] = a[i]
```

α**+1** **=** β has solution α**=**β**-1**

```
    for(i=0;i<N,i++)
S1    a[2*i+1] = a[2*i]
```

2*α**+1** **=** **2***β has no solution

General Cases

- Multiple dimensions → Multiple linear equations
- Loop limits constrain the loop index values
- Dependence directions also constrain the loop index values (for loop-carried dependences)
- Loop indexes have integer values
- Therefore, the general model is *integer linear programming*
  - Integer programming also applies to cases in which loop limits and subscripts contain linear expressions with non-index variables.
  - Many special-case linear systems exist, which can be solved fast.

The Issue of Symbolic Terms

```
  for(i=0;i<N,i++)
S1  a[f(i)] = a[g(i)]
```

- A *dependence equation* defines the access requirement

To prove **flow dependence**:
for which values of $\alpha < \beta$ is
f(α) = g(β)

```
  for(i=0;i<N,i++)
S1  a[i+p] = a[i+q]
```

p $\neq$ q ?

One way to find the answer is to substitute p and q with their definitions
This motivates SSA (static single assignment)

Privatizing variables

- In loops, especially loops in nonnumerical programs, many variables are used for intermediate results within each iteration
- In the parallel code, these can be
  - allocated to the thread stack (explicit threading code), or
  - marked as threadprivate or task-private (in OpenMP)

How to recognize privatizable variables

- Definition: upward-exposed reads
  - If a read reference to x always follows a write reference to x *in the same loop iteration, this read reference is said to be covered.*
  - If a reference is not covered, then it is said to be *upward-exposed.*

```
while continue {

 x = …;

If … y = x + x;
// read x is covered

}
```

```
while continue {

 if cond x = …;

If … y = x + x;
// read x is not covered

}
```

- Claim: If all read references to x in the loop are covered, then x can be *privatized* in that loop.
- Note: read-only variables do not need to be privatized to change parallelizability of the loop

Compiler analysis for privatizable variables

- For simple scalar variables, privatizability can be conservatively approximated using traditional compiler analyses for
  - reaching definitions, or use
  - variable liveness analysis

- Conventional compiler algorithms however do not take the meaning of the if conditions into account

```
while continue {
 …
If cond1 x = …;
 …
If cond2 y = x+x;

}
```

Does cond2 imply cond1?

Privatizable Arrays

- Arrays may also be used to hold intermediate results in a loop
- Analysis of privatizability of arrays requires an extension of the previous definition
- Definition:
  - If a read reference to an array element always follows a write reference to the same array element *in the same loop iteration,* this read reference is said to be *covered.*
  - If a reference is not covered, then it is said to be *upward-exposed.*

- Just like simply scalar variables, privatizability analysis can be sharpened by analyzing the meaning of IF conditions

lastprivate

- If in the original loop, the final value of a privatizable variable will be used after loop exits (the variable is live at the exit), then the final value must be *copied out*.
- Analyzing live scalar variable is a conventional compiler analysis
- Analyzing live array sections is a more advanced analysis
  - It again uses the concept of covered array sections
    - Any future references to any part of the array section being considered are upward exposed to the exits of the loop being considered.